Propositions and Types, Proofs and Programs

Part V: Extensions

Ranald Clouston
School of Computing
Australian National University

ANU Logic Summer School 2023

Gödel's System T

We mentioned previously that we would like our base types to include useful types such as Bool and Nat.

- Including introduction and elimination rules
- System T extends STLC with exactly these types

These types are not interesting from a traditional logical point of view

- Logically equivalent to each other, and to any other theorem, e.g. $\bot \to \bot$
- But logical equivalence does not imply isomorphism
- This is the **proof relevant** view of logic which cares about which proofs inhabit each type, not mere inhabitation.

Booleans

Introduction and elimination:

$$\frac{}{\Gamma \vdash \texttt{True:Bool}} \qquad \frac{}{\Gamma \vdash \texttt{False:Bool}}$$

$$\frac{\Gamma \vdash \texttt{b:Bool} \quad \Gamma \vdash \texttt{t:C} \quad \Gamma \vdash \texttt{u:C}}{\Gamma \vdash \texttt{if b then t else u:C}}$$

Reduction rules:

- `if True then t else u` $\mapsto$ `t` and `if False then t else u` $\mapsto$ `u`
- Perhaps further rules for subformula property.

Booleans Discussed

We can therefore write programs with Booleans like

- $\lambda x^{\mathtt{Bool}}.\mathtt{if}\ x\ \mathtt{then}\ \mathtt{False}\ \mathtt{else}\ \mathtt{True}$
- $\lambda x^{\mathtt{Bool}}.\lambda y^{\mathtt{Bool}}.\mathtt{if}\ x\ \mathtt{then}\ y\ \mathtt{else}\ \mathtt{False}$

We could instead encode Bool in STLC

- $(0\to 0)+(0\to 0)$
- True as $\iota^1(\lambda x^0.x)$ and False as $\iota^2(\lambda x^0.x)$
- if…then…else left as an exercise

But Nat, which is not finite, will require us to go beyond STLC

Natural Numbers

Introduction and elimination:

$$\frac{}{\Gamma \vdash \texttt{zero}:\texttt{Nat}} \qquad \frac{\Gamma \vdash \texttt{n}:\texttt{Nat}}{\Gamma \vdash \texttt{suc n}:\texttt{Nat}}$$

$$\frac{\Gamma \vdash \texttt{n}:\texttt{Nat} \quad \Gamma \vdash \texttt{t}:\texttt{A} \quad \Gamma \vdash \texttt{f}:\texttt{A}\rightarrow(\texttt{Nat}\rightarrow\texttt{A})}{\Gamma \vdash \texttt{rec(n,t,f)}:\texttt{A}}$$

`rec` corresponds to **primitive recursion**

- `n` is a counter; `t` is the base (zero) case; `f` is the recursive step

Natural Numbers - Examples

Test for zero:

$\lambda x^{Nat}.\mathtt{rec}(x,\ \mathtt{True},\ \lambda z^{Bool}.\lambda z'^{Nat}.\mathtt{False})$

Plus:

$\lambda x^{Nat}.\lambda y^{Nat}.\mathtt{rec}(x,\ y,\ \lambda z^{Nat}.\lambda z'^{Nat}.\mathtt{succ}\ z)$

Predecessor (with zero mapped to zero):

$\lambda x^{Nat}.\mathtt{rec}(x,\ \mathtt{zero},\ \lambda z^{Nat}.\lambda z'^{Nat}.z')$

Natural Numbers - Reduction

Beta rules (we will ignore other rules e.g. for the subformula property):

- `rec(zero,t,f)  ↦ t`
- `rec(suc n,t,f) ↦ (f (rec(n,t,f))) n`

Some examples from the last slide:

- `rec(zero,True,λz.λz’.False) ↦ True`
- `rec(suc n,True,λz.λz’.False)`
  `↦ ((λz.λz’.False)(rec(n,True,λz.λz’.False)))n ↦* False`
- `λx.rec(suc n,zero,λz.λz’.z’)`
  `↦ ((λz.λz’.z’)(rec(n,zero,λz.λz’.z’)))n ↦* n`

System T and Termination

We must give up at least one of the three:

- Strong normalisation
- Decidable type-checking
- Turing completeness (all terminating algorithms representable)

System T with `Nat` types is much more expressive than STLC but is likewise *not* Turing-complete.

Ackermann's function:

`A(0,n)` = `n+1`; `A(m+1,0)` = `A(m,1)`; `A(m+1,n+1)` = `A(m,A(m+1,n))`

- Expressible in the untyped lambda calculus but not in System T.

Extending System T to Turing Completeness

To make System T a universal model of computation we need to strengthen the notion of recursion, and give up strong normalisation.

This is a perfectly sensible thing to do for a programming language but destroys the formal relationship with logic.

- *All* types are inhabited by programs that fail to terminate;
- So the corresponding logic is inconsistent.
- However as long as we attempt to write terminating programs only, we can take inspiration from Curry-Howard even in a Turing complete language;
- Indeed this has been a major source of design ideas for real programming languages.

Dependency

In the STLC and System T there is one sort of variable:

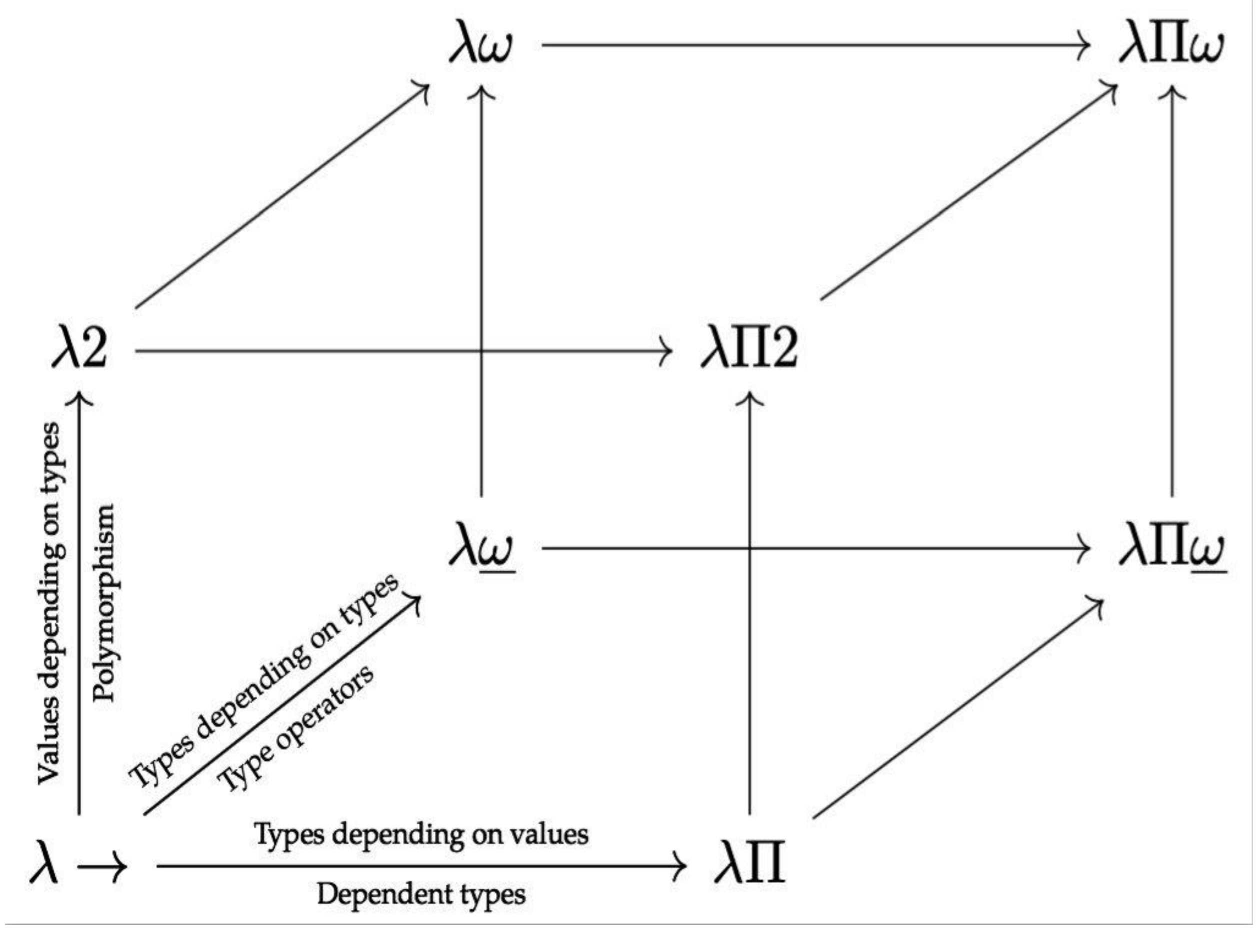
- Variables **appear in** terms; and can be **replaced by** terms
- We could say that **terms** depend on (unknown) **terms**.

But there are three other options that could be considered:

- Terms depend on types;
- Types depend on terms;
- Types depend on types

Each option supports different programming paradigms, and relates via Curry-Howard to different notions of logic.

The Lambda Cube



Indeed these three approaches to variables can be combined in various ways.

We will start with λ2 a.k.a. **System F**

- For the programmer, **parametric polymorphism**
- For the logician, **second-order propositional logic**

Diagram c/o [joomy on Twitter]

Types and Terms of System F

Given a set of **type variables** X,Y,…

- Type variables are types
- If A and B are types then so is A→B
- If A is a type then so is $\Pi X.A$. Note that Π binds its variable

Hence extend STLC with new term-formers, and typing rules:

$$\frac{\Gamma \vdash t:A}{\Gamma \vdash \Lambda X.t:\Pi X.A} \qquad \frac{\Gamma \vdash t:\Pi X.A}{\Gamma \vdash tB:A[B/X]}$$

The Λ rule is only permitted if X does not appear as a free variable in any type in Γ.

An Example

$\Pi X.X \to X$ is a closed type of `functions $X \to X$ for any X'

So it should be inhabited by a **polymorphic identity function**

- Clearly superior to individually defining functions for each type

$$\dfrac{\dfrac{\dfrac{}{x:X \vdash x:X}}{\vdash \lambda x^X.x : X \to X}}{\vdash \Lambda X.\lambda x^X.x : \Pi X.X \to X}$$

To turn it into an identity function that can be used at some specific type A we write

$$\vdash (\Lambda X.\lambda x^X.x)A : A \to A$$

Reduction for System F

Along with the usual beta-reduction we have

- (ΛX.t)A ↦ t[A/X]

The substitution will only effect the type annotations in terms

- Which we sometimes omit
- e.g. (ΛX.λx^{X}.x)A ↦ λx^{A}.x

All desirable properties proved for our weaker systems continue to hold

Note that real polymorphic languages e.g. Haskell tend to use type variables but not Λ

- Λ always implicitly at the outermost level

Encodings in System F

Chapter 11 of ‘Proofs and Types’ show that System F is powerful enough to encode many useful types

- Booleans, Products, Sums, Empty type;
- Existential Type (of which more soon);
- Natural numbers, Lists, Trees, inductively defined types more generally…

But polymorphism is not the only way to get access to the ability to define one’s own types

- The most natural approach is to develop STLC in another direction: ‘Types depend on types’
- In this setting e.g. Lists depend on their base type

Curry-Howard for System F

System F has an operator that binds variables at the type level

- As do all the extensions in the lambda cube, in fact

Via Curry-Howard, we would like to relate this to connectives that bind variables in propositions.

We indeed know two such connectives: ∀ and ∃

- So all the extensions in the lambda cube relate to some sort of quantifier:

Terms depend on types	Second order quantification
Types depend on terms	First order quantification
Types depend on types	Higher order quantification

Second Order ∀

Second order (propositional) logic has formulae ∀X.A where X is understood to be an **arbitrary proposition**

- *Not* an arbitrary element of some domain of elements – that is *first* order

So e.g. ∀X.X→X is a theorem

- 'For any proposition X we have X→X'

∀X.X∨¬X is not a theorem, but nor is it a contradiction

- We are still working in intuitionistic logic, albeit in powerful extensions!
- We could take this as an assumption to prove certain things

What About Second Order ∃?

Is there an analogue of second order ∃ that is useful for programming?

Yes!

- Existential types correspond to **abstract data types**.
- Symbol usually written Σ
- By asserting a type exists, but not committing to what it is, we can work (from the outside) with something without caring about its internal representation.

Example: heterogenous lists

- Usual polymorphic lists can only be instantiated to contain data of one type;
- But (assuming we have a polymorphic list constructor) `List(ΣX.X)` is a list of values that each have some type; each type might be different!

Quantifiers in Intuitionistic Logic

Stepping back to the BHK Interpretation:

- A proof of $\forall x:?.A$ is a construction that transforms a proof of $a \in ?$ into a proof of $A[a/x]$
- A proof of $\exists x:?.A$ is given by providing $a \in ?$, and a proof of $A[a/x]$

The ? depends on which sort of logic we are doing

- Left out when I want to be vague or it is clear from context
- Second order: ? is the set of propositions

So proofs of ∀ propositions are functions, and of ∃ propositions are pairs

- E.g. $\Sigma X.X$ is a pair of a type, and an element of that type
- Programming: usually written $\Pi x:?.A$ and $\Sigma x:?.A$
- But sometimes $(x:?)\to A$ and $(x:?)\times A$

Constructive ∃

Defining ∃ via pairs creates another difference with classical logic.

- $\exists x{:}A \leftrightarrow \neg(\forall x.\neg A)$ and $\forall x.A \leftrightarrow \neg(\exists x{:}\neg A)$ fail
- Analogous to failure of e.g. $A \wedge B \leftrightarrow \neg(\neg A \vee \neg B)$

The impossibility of a certain construction failing to exist does not intuitionistically imply that the construction does exist.

- We need to provide the element a, *then* prove it has property $A[a/x]$
- To prove an irrational number exists, it is *not* enough to prove that not all real numbers can be rational; an example of an irrational number must be given.
- This is the heart of **constructive proof**.
- A constructive proof of an existential can be used as an algorithm: it tells you how to generate the element that has the desired property.

Dependent Types

Types depend on terms

- Corresponds to **typed** first order logic
- We do not merely write $\forall x.B$, but instead $\forall x{:}A.B$
- So the elements used to instantiate variables are proofs / programs.

The proof / typing rules resemble those of propositional logic:

$$\frac{\Gamma,\ x{:}A \vdash t{:}B}{\Gamma \vdash \lambda x^{A}.t{:}\Pi x{:}A.B} \qquad \frac{\Gamma \vdash t{:}\ \Pi x{:}A.B \quad \Gamma \vdash u{:}A}{\Gamma \vdash tu{:}B[u/x]}$$

Dependent Types for Theorem Checking

Dependent types are the central technology behind many (not all!) **interactive theorem provers / proof assistants**

- Coq, Lean, Agda, Idris, etc…
- Dependent types key, but actually at top right of the lambda cube
- The most spectacular and useful application of Curry-Howard
- Types can express propositions (e.g. mathematically meaningful statements), and programs are proofs of them
- If a program **type-checks** (done automatically) then it is correct!
- **Machine-checked** (not usually machine-generated) mathematics.

Dependent Types in Action

Suppose we have the ingredients

- The type Nat and sum (disjunction, not addition) type former +
- For any type A and two terms t and u of type A, an identity type $t =_A u$. Note that this type depends on terms! The type $t =_A t$ is always inhabited; let's call its inhabitant $\texttt{refl}_t$.
- A type Type that contains all the 'small' types (not including e.g. itself) as elements, e.g. ⊢ Nat:Type
- The ability to define new types by recursion (we will use only primitive recursion so as not to break strong normalisation).

We will use these ingredients to prove the meaningful (if rather obvious) mathematical statement:

There is no largest natural number

Dependent Types in Action: >

First let us rephrase our statement without negation to give a more constructive proof:

- For every natural number n there exists a natural number m such that m > n
- We must give an algorithm that constructs this m *and* proves its property
- It should have type Πn:Nat.Σm:Nat.m > n

For this to work we need to define m > n as a (dependent) type

- *Not* an element of Bool – a type of proofs that m is greater than n!

Inductively:

- ⊢ _>_ : Πm:Nat.Πn:Nat.Type
- (succ m>n) := $(m =_{Nat} n)$+(m>n)

Dependent Types in Action

Given

- $(\mathtt{succ}\ m > n) := (m =_{Nat} n) + (m > n)$

we can prove $\mathtt{succ}\ m > m$:

- $m:\mathrm{Nat} \vdash \iota^1\ \mathtt{refl}_m : \mathtt{succ}\ m > m$

And can hence prove our target theorem:

$$\vdash \lambda^{Nat} x. \langle \mathtt{succ}\ x, \iota^1\ \mathtt{refl}_x \rangle : \Pi n{:}\mathrm{Nat}. \Sigma m{:}\mathrm{Nat}. m > n$$

So the Curry-Howard isomorphism has been used to prove a real statement of mathematics, in a style a computer could check!

Next steps: Exploring the Theory

- Proofs and Types by Girard
- Much more in-depth: Lectures on the Curry-Howard Isomorphism by Sørensen and Urzyczyn
- Follow citations from Propositions as Types by Wadler
- Look at some of my work on links between type theory and *modal* logic: Fitch-Style Modal Lambda-Calculi, or (more category theoretic) Modal Dependent Type Theory and Dependent Right Adjoints

Next steps: Curry-Howard in Practice

Get started with Coq, or Lean, or Agda, or Idris!

Multiple members of ANU's Foundations Cluster ready to supervise projects on theorem provers based on Curry-Howard

- As well as experts on theorem provers based on other ideas

In particular I am currently planning to run a small reading group on the online textbook Theorem Proving in Lean in the summer session (Jan-Mar)

- 6 units as COMP3740 or Advanced Studies Course

Any More Questions?

And…

Thanks for your time!